Grundlagen der Theoretischen Informatik

Sebastian Iwanowski
FH Wedel

Kap. 3: Grundlagen logischer Programmierung

Das Ziel logischer Programmierung

Konstruktionsaufgabe: *nicht berechenbar im Allgemeinen !*

Gegeben eine Menge $\mathscr{F}$ von prädikatenlogischen Formeln. Bestimme alle Formeln F, die aus $\mathscr{F}$ folgen.

Verifikationsaufgabe: *auch nicht berechenbar im Allgemeinen!*

Gegeben eine Menge $\mathscr{F}$ von prädikatenlogischer Formeln und eine (neue) prädikatenlogische Formel F. Berechne, ob F aus $\mathscr{F}$ folgt.

Weitere Möglichkeit der Vereinfachung des Problems:

Schränke die zulässigen Formeln ein!

Äquivalente Formulierungen zur Verifikationsaufgabe:

1) Gegeben eine Menge $\mathscr{F}$ von prädikatenlogischen Formeln und eine (neue) prädikatenlogische Formel F. Berechne, ob die Formelmenge $\{\neg F\} \cup \mathscr{F}$ widersprüchlich ist.
2) Gegeben eine Menge $\mathscr{F}$ von prädikatenlogischen Formeln. Berechne, ob sie widersprüchlich ist.

Das Ziel logischer Programmierung

Wdh.:

Die Prädikatenlogik 1. Stufe erweitert die Aussagenlogik um folgende Elemente:

- **Prädikate**
  - Aussagen, die von Variablen abhängen
- **Variable**
  - entsprechen den Literalen der Aussagenlogik, können aber beliebig viele Werte annehmen
- **Funktionen**
  - eindeutige Zuordnungen, die von Variablen abhängen
- **Quantoren**
  - Existenzquantor (∃) und Allquantor (∀)
  - Quantoren werden nur auf Variablen angewendet (sonst nicht 1. Stufe)

Die logische Programmierung versucht, die Verifikationsaufgabe für eine spezielle Klasse von Formeln zu erfüllen, die immer noch alle oben genannten Eigenschaften erfüllt.

Widerspruchsfindung mittels Resolution

Aufgabe:

Gegeben eine Menge $\mathcal{F}$ von prädikatenlogischen Formeln. Berechne, ob sie widersprüchlich ist.

Methode:

Äquivalente Formelumformungen: Ziel ist es, die Konstante $\perp$ herzuleiten.

Wir arbeiten so lange mit allgemeinen Formeln, bis es nicht mehr weitergeht!

Resolutionsprinzip:

Generierung einer neuen Formel als Folgerung aus 2 gegebenen Formeln

Prinzip: Finde Literal c, der in den Formeln a ∨ c und b ∨ ¬c vorkommt.

Dann kann c **eleminiert** werden: $(a \vee c) \wedge (b \vee \neg c) \rightarrow (a \vee b)$

Die neue Formel heißt **Resolvente** der alten Formeln.

Durch eine solche Eliminierung können einzelne Literale isoliert werden:

Bsp.: $(a \vee c) \wedge \neg c \rightarrow a$ Interpretation: a muss in der Formelsammlung gelten.

Wenn auf diese Weise auch die Negation isoliert wird, ergibt sich ein Widerspruch: ***Widerspruch!***

Bsp.: $(\neg a \vee d) \wedge \neg d \rightarrow \neg a$ Interpretation: ¬a muss in der Formelsammlung gelten.

Reduktion der Termvielfalt mittels Unifikation

Beispiel:

$$\Phi = \{\neg P(x, f(y)) , P(z, f(g(z)))\}$$

Frage: Wieso kann diese Formelmenge widersprüchlich sein ?

Antwort: Weil die beiden Atome $P(x, f(y))$ und $P(z, f(g(z)))$
durch geschickte Wahl von z und g identifiziert werden können.

Das kann man mit Resolution alleine nicht herausfinden!

Eine logische Programmiersprache braucht daher *Unifikation*:

Ersetzung der Variablen durch Terme, so dass beide Atome gleich werden.

Reduktion der Termvielfalt mittels Unifikation

Substitution:

Die Ersetzung [*x*/*t*] angewendet auf φ bezeichnet diejenige Formel, die aus φ entsteht, wenn alle ***freien*** Vorkommen in φ von *x* durch Term *t* ersetzt werden.
Analog wird die Ersetzung [x1/t1, x2/t2, ..., xn/tn] gebildet.

Bezeichnung: σ = [x1/t1, x2/t2, ..., xn/tn] heißt ***Substitution***.
$\sigma\,\varphi$ *ist* die ***Anwendung von*** σ auf φ

Beispiel: Formel: φ = P (f (x), y)
Substitution: σ = [x/z, y/f (z)]
Anwendung: $\sigma\,\varphi$ = P (f (z), f (z))

Definition:

Eine Substitution σ heißt ***Unifikator*** für die Formeln α_1 und α_2, wenn gilt: $\sigma\alpha_1 = \sigma\alpha_2$.

Beispiel:

Unifikation der Atome *Q* (*f* (*x*), *v*, *b*) und *Q* (*f* (*a*), *g* (*u*), *y*)
durch die Substitution σ = [*x*/*a*,*v*/*g*(*u*), *y*/*b*]

Reduktion der Termvielfalt mittels Unifikation

Satz (Existenz):

Für je zwei Ausdrücke gibt es, bis auf Variablenumbenennung, entweder einen eindeutigen allgemeinsten Unifikator oder die beiden Ausdrücke sind nicht unifizierbar.

Satz (Berechenbarkeit):

Es gibt einen Algorithmus, der für zwei beliebige Ausdrücke entweder die Nichtunifizierbarkeit beweist oder den allgemeinsten Unifikator berechnet.

Verfahren: *ziemlich einfach !*

Wiederhole bis die Ausdrücke gleich sind oder die Nichtunifizierbarkeit gezeigt ist:
- **Wenn** die Prädikate verschieden sind
  - oder die Anzahl der Parameter gleicher Prädikate verschieden ist
    - → nicht unifizierbar
- **Anderenfalls**
  - Wähle eine Variable x im ersten Ausdruck
    - und einen an der äquivalenten Stelle stehenden Term t im zweiten Ausdruck, der x nicht enthält.
  - **Wenn** das nicht möglich ist → nicht unifizierbar
  - **Anderenfalls** ersetze x in beiden Ausdrücken durch t.

Reduktion der Termvielfalt mittels Unifikation

Übungsbeispiele für Unifikation:

$P(x)$ und $Q(y)$

$P(x, y)$ und $P(z)$

$P(x, y)$ und $P(a, f(a))$

$P(x, y)$ und $P(f(z), g(z))$

$P(x, f(x, x), z, f(z, z))$ und $P(f(a, a), y, f(y, y), u)$

$P(x, f(y))$ und $P(z, f(g(z))$

$P(x, x)$ und $P(f(y), f(g(z))$

$P(x, f(x))$ und $P(y, y)$

$P(x, a)$ und $P(b, x)$

Logisches Programmieren

Das Prinzip der Programmiersprache PROLOG:

PROLOG versucht, mittels wiederholter und verschachtelter Anwendung
von **Resolution** und **Unifikation**
zu einer gegebenen Formelmenge einen Widerspruch zu finden.

Satz (Widerspruchsvollständigkeit):

Falls die Formelmenge widersprüchlich ist, kann man den Widerspruch immer finden.

Was fehlt ?

Satz (*Folgerung* der Folgerbarkeit aus der Widerspruchsaufdeckung):

Wer zu jeder Formelmenge jeden Widerspruch aufdecken kann,
kann zu jeder Formelmenge und zu jeder neuen ***daraus folgenden*** Formel beweisen,
dass die neue Formel aus der alten Formelmenge folgt.

Was fehlt hier ?

Logisches Programmieren

Wie macht man aus PROLOG eine <u>vollständige</u> Programmiersprache ?

Durch Beschränkung der Eingabe !

PROLOG akzeptiert nur Mengen von Formeln der Form:

p ∧ q ∧ . . . ∧ r → x

Regeln (Hornklauseln)

In der Voraussetzung darf nur eine Konjunktion von positiven Literalen stehen.

<u>Satz (Vollständigkeit der Resolution auf Hornklauseln):</u>

Für jede Menge von Hornklauseln und eine neue Hornklausel kann Prolog nach endlicher Zeit entscheiden, ob die neue Hornklausel aus der alten Menge folgt ***oder nicht***

☹ ***Anmerkung: „Endliche Zeit“ kann „sehr lange“ heißen !***

Logisches Programmieren

Wie erkennt man, ob eine Formelmenge nur aus Formeln besteht, die äquivalent zu Hornklauseln sind ?

Die Formelmenge sei in KNF gegeben, d.h. als eine Konjunktion von Disjunktionen.
Die einzelnen Klauseln (Disjunktionen) seien die einzelnen Formeln.
Eine einzelne Formel ist nach Definition eine Hornklausel, wenn sie äquivalent zu einer Regel ist, deren Voraussetzungen alle durch einen positiven Literal repräsentiert sind.

Eine Hornklausel sieht also immer so aus:

$$\neg p \vee \neg q \vee \ldots \vee \neg r \vee x$$ *Maximal ein Literal ist positiv.*

Anmerkung:
Natürlich kann man eine beliebige Aussage auch durch einen negativen Literal repräsentieren.
Um die Hornklauseleigenschaft zu erhalten, muss dieser Literal dann aber in allen Voraussetzungen von Regeln, in denen er vorkommt, in negativer Form auftauchen.
Daher können wir uns ohne Beschränkung der Allgemeinheit auf positive Literale beschränken.